Carnegie Mellon
Software Engineering Institute

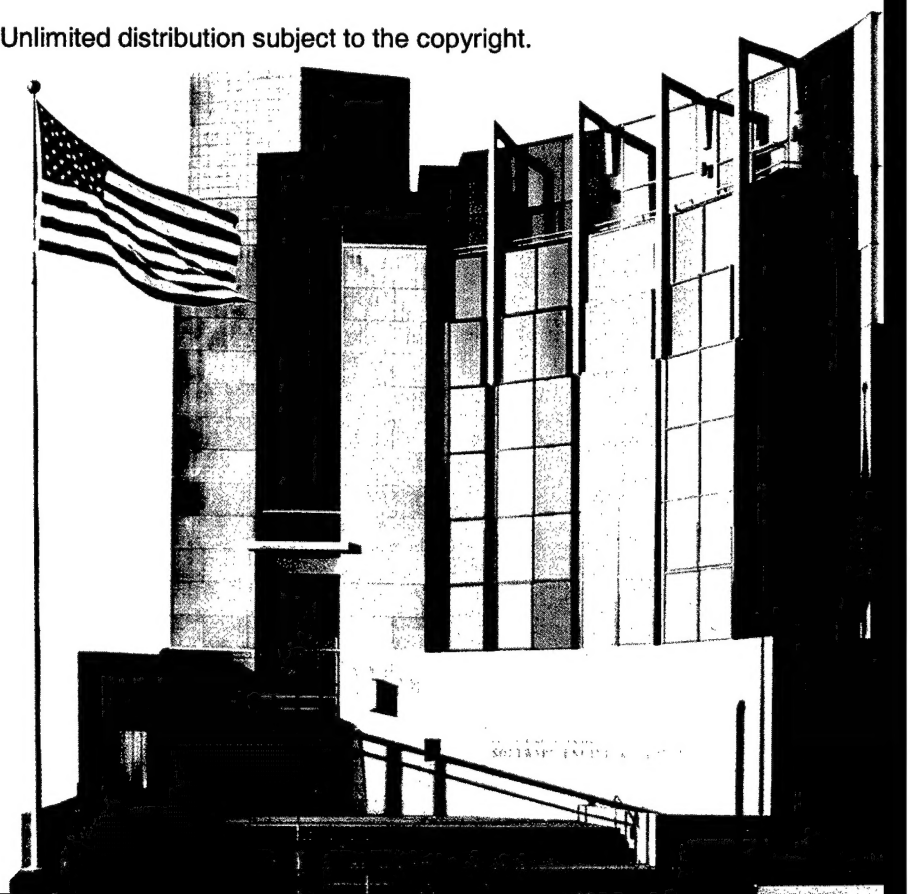
Applying FSQ Engineering Foundations to Automated Calculation of Program Behavior

Richard C. Linger

February 2003

Network Systems Survivability

Technical Note
CMU/SEI-2003-TN-003



Applying FSQ Engineering Foundations to Automated Calculation of Program Behavior

Richard C. Linger

February 2003

Network Systems Survivability

Technical Note
CMU/SEI-2003-TN-003

Contents

List of Figures

Abstract

No software engineer can say with assurance how a sizable program, with its virtually infinite number of possible execution paths, will behave, that is, what it will do, in all circumstances of use. This incredible reality, widely acknowledged but little discussed, lies at the heart of intractable problems experienced in software development and use over the past 40 years. If full behavior is unknown, so too are embedded errors, vulnerabilities, and malicious code that can emerge in use. While this reality has seemed inevitable in the past, it need not be so in the future. The SEI CERT Coordination Center has been conducting research on Flow-Service-Quality (FSQ) engineering for complex, network-centric system analysis and development. FSQ Flow Structures treat the control structures of programs as rules, or implementations, of mathematical functions, that is, mappings from domains to ranges. The function, or behavior, of any control structure can be abstracted into a procedure-free statement that specifies its net functional effect in all circumstances of use with mathematical precision. The finite number of control structures in a program can be abstracted in stepwise fashion in an algebra of functions, to arrive at a precise statement of the program's overall behavior. The mathematical foundations largely exist, and development of such a capability is feasible, albeit difficult. Automated program behavior calculation would have a dramatic effect on software and systems engineering, and enable a new level of assurance in trustworthy systems. This report briefly summarizes research to date on Flow Structures and describes the application of their function-theoretic mathematical foundations to the problem of program behavior calculation.

1 The Problem of Understanding Program Behavior

No software engineer can say for sure what a sizable computer program does in all circumstances of use. Yet incredibly, modern society is totally dependent on the correct functioning of countless large-scale systems composed of programs whose full behavior and fitness for use are not reliably known. It is little wonder that system development is a risky and unpredictable proposition, and that systems experience an endless flood of unforeseen bugs, vulnerabilities, and malicious code with frequently serious consequences. Failures in system development have been recently estimated to waste a quarter trillion dollars per year [Morgan 2002]. Such a situation in other engineering disciplines would not be tolerated. The state of affairs is illuminated by a principle argument of the open source software movement that maintains that more people looking at program code will find more errors. It is interesting to observe that there is no open source arithmetic movement, seeking more people to determine if sums are correct. Society knows how to make sums correct and has automated the process. It turns out that the same can be true of software.

The task of program understanding today is a haphazard and error-prone process carried out by programmers in human time scale. Because understanding of behavior is an essential prerequisite to effective program development and modification, programmers are forced to devote substantial time to this task. Reliable understanding is also essential for discovery of errors, vulnerabilities, and malicious code. Compounding the problem is the difficulty of understanding programs written by others. And because unscrupulous programmers and intruders can make deleterious modifications to programs at any time, the task of behavior discovery never ends.

Why are sizable programs so hard to understand? It is because they contain a virtually infinite number of possible execution paths, any of which may be relevant to the development or modification task at hand, and any of which may contain errors, vulnerabilities, or malicious code. Faced with massive sets of possible executions, programmers, constrained by limits on time and concentration, typically focus on gaining a general understanding of mainline program behavior. There is simply no way to understand and remember it all in today's state of the art.

While this problem has seemed intractable in the past, it may not be so in the future. The mathematical foundations of software illuminate a difficult but feasible strategy to develop new types of automation that can address the problem of program understanding in an innovative way. These possibilities stem from function-theoretic mathematical semantics that have been applied in the Flow-Service-Quality engineering project carried out by the CERT Coordination Center, as well as from extensions to the semantics defined in that project. The opportunity exists to move from an incomplete understanding of program behavior

laboriously derived in human time scale to a precise calculation of program behavior automatically derived in CPU time scale.

The key to the function-theoretic approach is the recognition that, while programs may contain a virtually infinite number of execution paths, they are at the same time composed of a finite number of control structures. It is this finite nature of program logic viewed through the lens of function theory that opens the possibility of automated calculation of program behavior.

2 Background: Function-Theoretic Foundations of FSQ Flow Structures

Research work on Flow-Service-Quality engineering has been documented in other reports [Linger 2002].[1] Flow Structures are a key element of FSQ engineering. They provide stable engineering foundations for analysis and development of dynamic, network-centric systems of systems that are characterized by unpredictable boundaries, uncertain function and quality of commercial off-the-shelf (COTS) components, and limited control of security and survivability domains. The following discussion briefly summarizes research to date on Flow Structures and describes application of their mathematical foundations to the problem of program behavior calculation.

Flow Structures are a representation and reasoning framework for specifying user task flows and their precise refinements into uses of system services in traversing a system architecture [Hevner 2001, Hevner 2002]. System services include all the functional capabilities of a system, from operating systems and communication protocols, to middleware and applications, to operations carried out by users and administrators. System services may be provided by local system components that are well understood and trusted, by COTS components of potentially uncertain function and quality, or by External Service Provider (ESP) components for which even less information may be available on functionality and quality of service. Architecture traversals by flows may visit and compose many computation and communication hardware and software components distributed across multiple systems. Mission-critical operations within an enterprise are ultimately carried out by user task flows that define the sequencing and composition of system services provided by these components to satisfy mission objectives. Survivability of these essential flows in adverse environments of intrusion and compromise is a requirement for mission continuity.

Flow Structures invoke system services that may be engaged in simultaneous and asynchronous use by other flows. However, a new approach to flow semantics permits flows themselves to be deterministic, despite the underlying asynchronous behavior of their constituent services. The basic FSQ semantic model of Flow Structures is the well-known functional model [Hoffman 2001, Mills 1986, Mills 2002, Prowell 1999] that treats programs as rules for mathematical functions, that is, mappings from domains (inputs, stimuli) to ranges (outputs, responses). This model can be extended to a new semantics that permits flows to be defined as deterministic entities, no matter what changing or unpredictable behavior is exhibited by their constituent services [Linger 2002]. This result permits flows to

[1] Also Linger, R. *Essential Service and Sense-and-Respond Control Models* (CMU/SEI-2002-SR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

be abstracted, refined, and verified with precision. Deterministic flows can be represented in straightforward single-entry, single-exit sequence (composition), alternation (ifthenelse), and iteration (whiledo) control structures (plus their variants and extensions) for human reasoning and analysis. Concurrent structures can be incorporated into flows as well.

Flow Structure engineering requires that, for survivability, flow designs deal explicitly with uncertainty factors characteristic of large-scale, network-centric systems, including unpredictable functionality and reliability of COTS and ESP components, as well as essential functionality that may be damaged or compromised at any time. This requirement supports both enterprise risk management and survivability engineering. In concrete terms, it requires that critical service invocations in flows be combined with post-fix predicates on subject-matter-dependent, designer-defined equivalence classes on all possible service responses. For example, use of a radar service to obtain an aircraft position fix could be followed by predicates on equivalence classes to determine whether (*a*) a response was provided (existence of the service), (*b*) whether the response is a position fix (potentially correct response), and (*c*) whether the position fix is, say, valid in comparison to the previous fix (presumed correct response). It is up to flow designers to select such critical services for response analysis. Sensing and responding to all possible outcomes in this manner is the essence of survivability engineering, which requires that systems take appropriate actions under all conditions of use, whether benign or adverse, expected or unexpected. In short, Flow Structures require sensing adverse events and responding correctly to them.

Discussion of the mathematical semantics and engineering operations associated with Flow Structures can be found in recent reports [Linger 2002].[2] The foundations of Flow Structures are expressed in a number of theorems, including the following:

- Flow Structure Theorem

  The Flow Structure theorem guarantees the sufficiency of sequence, alternation, and iteration control structures to represent any sequential flow. (Extensions and variants of these structures are included as well.) Thus, flows can be expressed in nested and sequenced single-entry, single-exit structures, each with a common underlying mathematical model, namely, a function mapping from domain to range.

- Abstraction/Refinement Theorem

  The Abstraction/Refinement theorem addresses conditions for substitution of flow specifications and their refinements, thus enabling precise abstraction, refinement, and verification operations.

[2] Also Linger, R. *Essential Service and Sense-and-Respond Control Models* (CMU/SEI-2002-SR-004). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002.

- Flow Verification Theorem

  The Flow Verification theorem defines conditions for correctness of flow control structures with respect to their specifications. Even though flows can contain a virtually infinite number of paths from start to end, they are expressed in a finite number of control structures, each of which can be verified in team inspections in from one to three reasoning steps as defined by the theorem. Thus verification is reduced to a finite and practical process.

These theorems provide foundations for the engineering operations of refinement, abstraction, and verification of Flow Structures. These are the very operations required in program development, modification, and quality assurance, regardless of the programming language or subject matter involved. In particular, precise abstraction is the key to program understanding, as discussed next.

3 Function-Theoretic Calculation of Program Behavior

The foundations of Flow Structures summarized above have natural application to the problem of program understanding, as a basis for automated calculation of the full behavior of programs in support of software and systems engineering. The key to intellectual control and management of software development, modification, and evolution is capturing and understanding the full functional effect of programs with mathematical precision. It is taken for granted that complete understanding of expressions is achievable and essential in other mathematical disciplines, but it remains an elusive goal in software today.

Program behavior abstraction can be accomplished through the function-theoretic mathematics and methods [Hausler 1990, Hoffman 2001, Mills 1986, Mills 2002, Parnas 1994, Pleszkoch 1990, Prowell 1999] that have been previously applied to the development of FSQ engineering concepts in this project. As noted, programs are compact representations of very large sets of possible behaviors. The process of deriving and expressing the net functional effect of program procedures in precise, non-procedural representations is known as *program abstraction*. The possibility of program abstraction arises from the recognition that programs and their constituent control structures implement mathematical functions. In the abstraction process, these functions are termed *program functions*. In informal illustration, the control structure operating on integers x, y, and z

```
if
   x > y
then
   z := x
else
   z := y
endif
```

can be abstracted to a procedure-free program function that can be expressed as

$$z := \max(x, y)$$

where the net effect of the structure is expressed in a single assignment of initial values of x and y to the final value of z, and x and y are unchanged. The canonical forms of the program functions of the basic control structures can be expressed through operations of function composition and case analysis as follows (for control structure labeled P, operations on data labeled g and h, predicate labeled p, and program function labeled f):

- Sequence control structure

  The program function of a sequence

  P: g; h

  can be given by

  f = [P] = [g; h] = [h] o [g]

  where the square brackets denote the program function of the enclosed program and "o" denotes the composition operator. That is, the program function of a sequence can be calculated by ordinary function composition of its constituent parts.

- Alternation control structure

  The program function of an alternation control structure

  P: if p then g else h endif

  can be given by

  f = [P] = [if p then g else h endif]

  = ([p] = true → [g] | [p] = false → [h])

  where | is the "or" symbol. That is, the program function of an alternation is given by a case analysis of the true and false branches, and the opportunity to combine them into a single abstraction as in the max illustration above.

- Iteration control structure

  For iteration control structures, the program function is given by function composition and case analysis in a recursive equation based on the equivalence of an iteration control structure and an iteration-free control structure (an ifthen structure):

  P: while p do g enddo

  can be reexpressed as

  f = [P] = [while p do g enddo]

  = [if p then g; while p do g enddo endif]

  = [if p then g; f endif]

  Function f is therefore given by

  f = ([P] = ([p] = true → [f] o [g] | [p]= false → I)

  where I is the identity function. Thus, the abstraction of an iteration structure is given by a two-part conditional rule, and the composition of f and g must be determined to define the true case.

A miniature example involving a sequence control structure can provide a notional sense of the operations involved in one low-level abstraction step in the behavior calculation process. Consider the sequence structure below composed of assignments that operate on small

integers x and y (matters of machine precision are left aside for the moment). The abstraction question asks: What does this program do, that is, what function does it compute? The answer is not obvious at first glance:

```
do
  x := x – y;
  y := y + x;
  x := y – x;
  y := y + abs(x);
enddo
```

Imagine a programmer writing this sequence and wondering if it does what is intended. An abstractor is invoked and the precise answer obtained in CPU time scale before continuing. Abstraction with mathematical precision requires deriving a procedure-free expression of what this structure does from beginning to end for all values of x and y. For a sequence structure, this requires composing the statements to determine their net, sequence-free effect. A trace table can be used for this purpose, with a row for every assignment statement and a column for every data variable assigned. Each cell in the table records the effect of the row assignment on the variables. Subscripts are attached to variables in the cells to index the effects from row to row, starting with 0 in the first row:

operation	x	y
x := x – y	x1 = x0 – y0	y1 = y0
y := y + x	x2 = x1	y2 = y1 + x1
x := y – x	x3 = y2 – x2	y3 = y2
y := y + abs(x)	x4 = x3	y4 = y3 + abs(x3)

The derivations below express the final values in the table in terms of the initial values through algebraic substitution:

```
x4 = x3
   = y2 - x2
   = y1 + x1 - x1
   = y1
   = y0
```

$$
\begin{aligned}
y4 &= y3 + abs(x3) \\
&= y2 + abs(y2 - x2) \\
&= y1 + x1 + abs(y1 + x1 - x1) \\
&= y0 + x0 - y0 + abs(y0) \\
&= x0 + abs(y0)
\end{aligned}
$$

Thus, the final value of x is the initial value of y, and the final value of y is initial value of x plus the absolute value of the initial value of y. That is, this control structure exchanges the values of x and y, and adds the absolute value of y to the latter. This abstracted function can be written as a concurrent assignment, in which initial values on the right are simultaneously assigned in order to final values on the left:

$$x, y := y, x + abs(y)$$

The control structure in this example is a simple sequence of operations whose behavior was derived in a straightforward trace table composition: alternation and iteration structures require trace tables that incorporate columns for conditions (predicates) as well, and the derivation of their final values in terms of initial values in similar fashion.

This behavior function precisely defines the net effect of the sequence, with matters of machine precision aside. If necessary, however, the finite nature of machine precision can be integrated into the analysis. For example, the properties of overflow and underflow can be dealt with in several referentially transparent ways, with the best approaches ultimately determined through experience with abstraction technology and user preferences:

1. Overflow and underflow can be ignored in the abstraction process. This corresponds to performing referentially transparent abstraction on a program and machine model that has infinite precision. In this case, the behavior function precisely defines the net effect of the sequence, with machine precision not accounted for, and is sufficient for many analytical purposes. It is the obvious choice where machine precision has no effect on particular operations. An advantage of this approach is that the overall program function is not obscured by details of finite precision; however, any behavior resulting from finite precision is lost. For example, consider the following simple exchange of integers, with attached program function in square brackets:

   ```
   [x, y := y, x]
   do
      x := x + y;
      y := x - y;
      x := x - y;
   enddo
   ```

   Note that in this case, if two's complement arithmetic is preformed, and overflow and underflow do not cause machine traps, then the exchange behavior is always correct,

even when overflow does occur, because the result must be correct modulo the word size of the executing machine. If necessary, however, the finite nature of machine precision can be integrated into behavior abstraction using one of the following approaches.

2. The domain of each potential overflow or underflow can be explicitly incorporated into the conditions of the conditional assignment statement. The finite nature of integer representations on a given machine introduces the possibility of underflow and overflow into the functional effect of the sequence, and the opportunity to produce other than the intended result. This corresponds to performing referentially transparent abstraction on a program and machine model with finite precision to a behavior model with infinite precision. This possibility can be accounted for by partitioning the domain and range of each assignment in the sequence into equivalence class regions, based in this case on subsets of initial values of x and y, within each of which the same functional results will be obtained. Some classes will produce the program function derived above, others will not. Incorporation of the operational semantics of machines is important for analysis of programs for vulnerabilities and malicious code intended to exploit, for example, finite properties and overflow characteristics of number representations or data structures such as buffers or registers. When the behavior calculations are augmented by operational semantics, such problems become obvious, with no additional analysis on the part of the user required. For example, consider the following program function for a single assignment:

```
[ ((x + y) >= 2^31) → overflow occurs
| ((x + y) < -2^31)  → negative overflow occurs
| true               → z := x + y ]
do
   z := x + y
enddo
```

An advantage of this approach is that the complete behavior of the program is captured in the behavior specification; however, the overflow and underflow conditions can obscure the primary logic of the program. This disadvantage can be mitigated by introducing variable bounds as preconditions and treating the behavior outside those preconditions as undefined. For example:

```
[ (abs(x) < 10^8) and (abs(y) < 10^8) → z := x + y
| true                                → undefined ]
do
   z := x + y
enddo
```

3. A third approach is to incorporate the operational semantics of the executing machine into the behavior calculation and simplification process as part of the trace table analysis. This corresponds to performing referentially transparent abstraction where the program and machine model, and the behavior model, are finite precision. That is, arithmetic operations in the behavior specification are subject to the same overflow and

underflow as in the program. For example, consider the following treatment of the exchange program:

```
[x, y := (x + y) - ((x + y) - y), (x + y) - y]
do
   x := x + y;
   y := x - y;
   x := x - y;
enddo
```

In this approach, "((x + y) - y)" cannot always be simplified to "x", because the original expression can have overflow, while the simplified expression cannot. A disadvantage of this approach is that overflow and underflow semantics are buried in behavior abstractions just as deeply as in the program statements. Behaviors that do not require simplification, however, will more clearly reflect the primary logic of the program.

These examples illustrate the power of the function-theoretic approach to deal with any behavioral and operational semantics appropriate to the problem at hand. As work on abstraction technology progresses, suitable vocabulary, definitions, reduction and simplification rules, and flexible user interfaces will emerge to support human preferences and understanding. In any case, it is important to recognize that the abstraction process is capable of extracting the true and complete behavior of any program or program part, the very behavior that exposes unforeseen errors and that intruders attempt to subvert for their own purposes. These behaviors are generated in the programmed functional logic and in its interaction with executing machines, and function-theoretic abstraction can deal completely and correctly with both.

Consider next behavior calculation for larger programs. The nested and sequenced control structures (sequence, ifthenelse, whiledo, etc.) in a program form an expression in an algebra of functions, where every control structure is a rule for a function as described above. In particular, the abstracted function defined by a given control structure can be freely substituted for the control structure itself, with no change in the meaning of the overall program, as summarized in an Axiom of Replacement. The control structures of a program define a natural decomposition hierarchy, wherein leaf node control structures can be abstracted into their program functions, thereby revealing new control structures now ready for abstraction, etc., continuing in this manner until the entire program has been abstracted into a single program function representing its net functional effect. At this point, all the procedural logic and local variables have been abstracted out, but their overall effect has been preserved in the final abstraction.

As illustrated in the example above, the abstracted program functions of control structures are conveniently recorded as a single statement in a closed specification language composed of a procedure-free concurrent assignment statement with general syntax

<id>, <id>, ..., <id> := <expr>, <expr>, ..., <expr>

(where <id> represents an identified data item, <expr> represents an expression that calculates a value for a data item, and the expressions on the right are simultaneously assigned in order to the data items on the left), and a concurrent conditional assignment statement with syntax

(<condition> → <concurrent assignment>

| <condition> → <concurrent assignment>

...

| <condition> → <concurrent assignment>)

where "|" represents "or" and <condition> is a truth-valued predicate. Any sequential logic can be abstracted into a program function expressed in this form.

The semantics of conditional concurrent assignments are easily represented in various graphic-based formats in a user interface for ready understanding and analysis.

In illustration of the stepwise abstraction process, consider the miniature program of Figure 1 and the question of what it does. The program is expressed in terms of a design language syntax, and is composed of sequence, ifthenelse, and whiledo control structures. It takes as input and produces as output a queue of integers named Q, and defines local queues of integers named odds and evens and a local integer variable named x.

The control structures of the program form a natural hierarchy with a number of leaf nodes. To begin the stepwise abstraction process, the lowest-level, leaf-node ifthenelse and sequence control structures of the program can be abstracted into non-procedural conditional concurrent assignments, as shown in Figure 2.

```
PROC (Q)
  odds, evens: queue of integer,
               initial empty
  x: integer
  WHILE Q <> empty
  DO
     x := end(Q)

     IF odd(x)
        THEN
          end(odds) := x
        ELSE
          end(evens) := x
     ENDIF
  ENDDO

  WHILE odds <> empty
  DO
     x := end(odds)
     end(Q) := x
  ENDDO

  WHILE evens <> empty
  DO
     x := end(evens)
     end(Q) := x
  ENDDO
ENDPROC
```

Figure 1: A Miniature Program for Abstraction

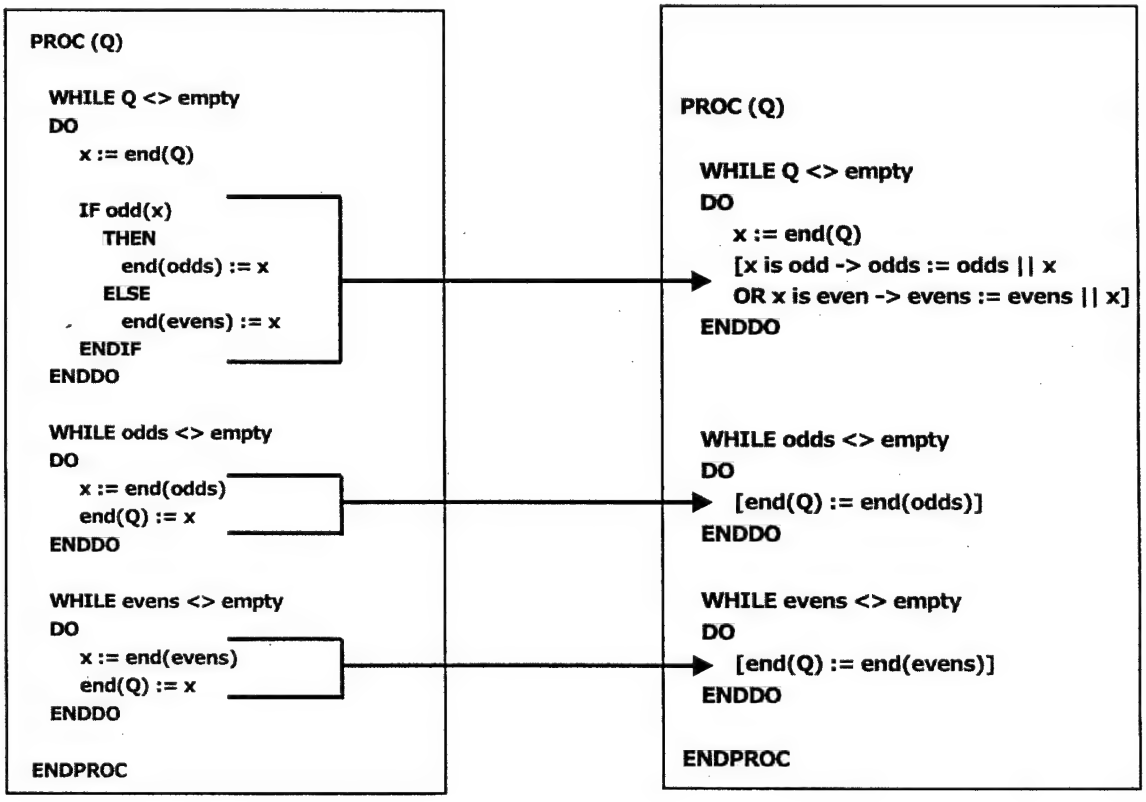


Figure 2: The First Abstraction Step

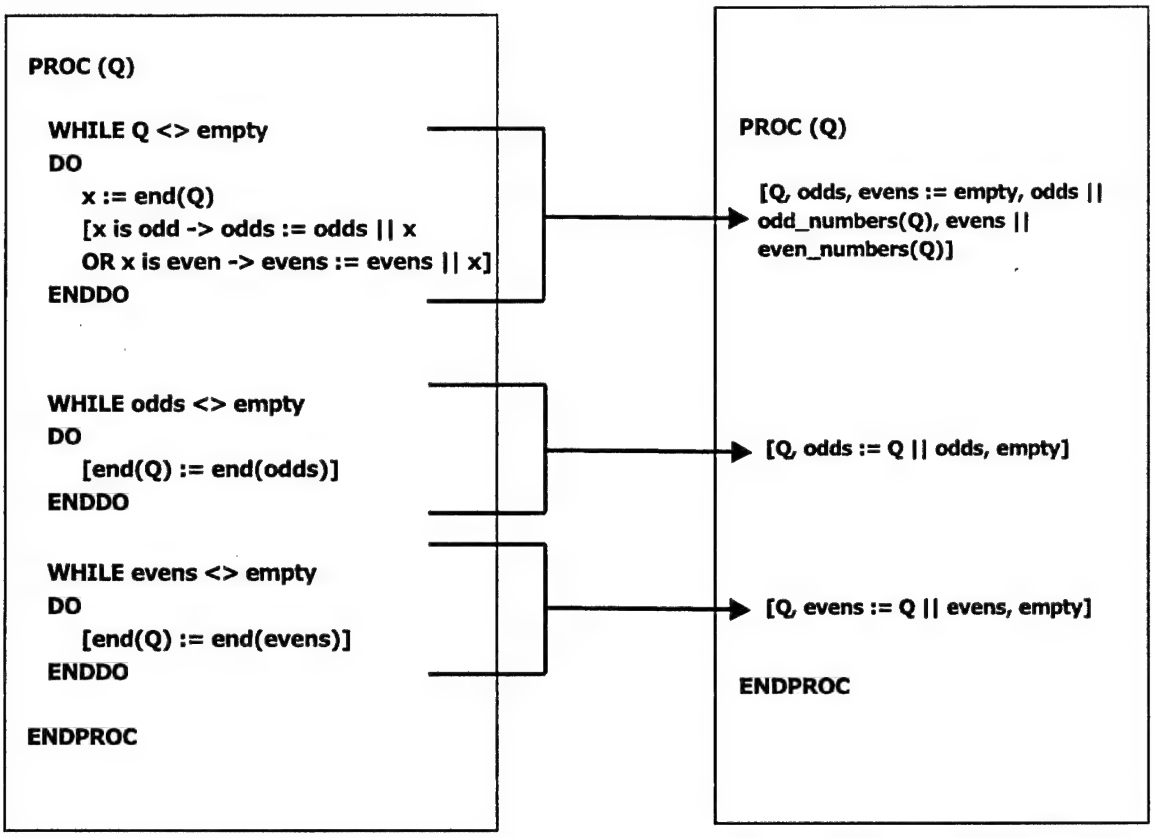


Figure 3: The Second Abstraction Step

Next, the three whiledo structures can likewise be abstracted to conditional rules and assignments, resulting in the program of Figure 3. Finally, the sequence of three abstractions can be composed into a single assignment expressing the overall behavior of the program as shown in Figure 4. This assignment precisely defines what the program does in functional terms. It is the as-built behavior specification, that is, the program function, of the program.

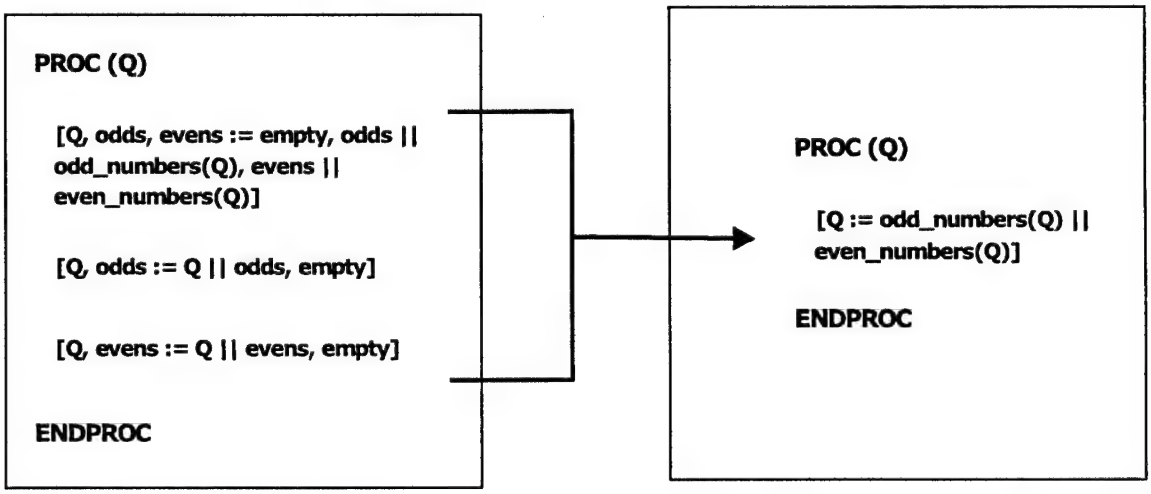


Figure 4: The Final Abstraction Step

Note in this process that intermediate control structures and data references drop out to simplify scale-up by subsuming their functional effects into higher-level abstractions. The principal behavior calculation process is function composition through value substitution, which by definition eliminates intermediate expressions at successive levels of abstraction. As noted above, programs can exhibit an enormous number of execution paths, but are comprised of a finite number of control structures, so the abstraction process is itself finite and guaranteed to terminate. Furthermore, behavior is recorded at each abstraction step, to produce complete functional documentation for human understanding at all levels.

This miniature example illustrates in informal terms a stepwise abstraction process that is invariant with respect to scale—the same mathematics and operations are employed at all levels of abstraction, no matter the size of the program. Were this program embedded in a larger system, it is its abstracted function that would participate in further abstraction, and not the program itself. In this way, local details are left behind at each step with no loss of information while precise abstractions propagate to higher levels. Abstraction does not mean vagueness; abstractions embody the precise net effect of implementation details. This process, combined with other techniques, limits complexity in behavior abstraction of large programs. Additional mathematical methods for unification and reduction must be brought to bear to simplify intermediate expressions and maintain intellectual control. These methods address the question of scale up to a practical industrial process, and are key elements of the required work to automate the process.

Larger programs are capable of more extensive behavior in mapping their inputs into outputs. Abstracted behavior of these programs is more extensive, and can be usefully organized into behavior catalogs. These catalogs are repositories of program behavior expressed in lists of conditional concurrent assignments and indexed according to the predicate expressions involved. Catalogs can be searched, browsed, and analyzed according to users' needs and objectives in investigating what a program does.

It is important to note that conditional concurrent assignments are engineering expressions of behavior that can be presented to users in forms that exhibit suitable human factors (graphical forms, in particular) as part of a familiar windowed user interface.

Consider next the problem of understanding the behavior of the Java program of Figure 5, which appears to perform a financial calculation. The program is shown on the left and its abstracted behavior catalog on the right. The abstraction was carried out through manual application of a behavior calculation algorithm. The calculated abstraction shown in its behavior catalog reveals non-trivial behavior summarized into three cases, expressed as conditional concurrent assignment statements. Each case begins with the conditions (predicate values) under which the corresponding transformations of data from input to output values will occur. The lists of data assignments within the cases occur simultaneously, but are shown in sequence for readability. In particular, the third case reveals complexities

that would be difficult and time-consuming indeed for fallible human analysis to uncover. It is simply not obvious that the program carries out the behavior cataloged here.

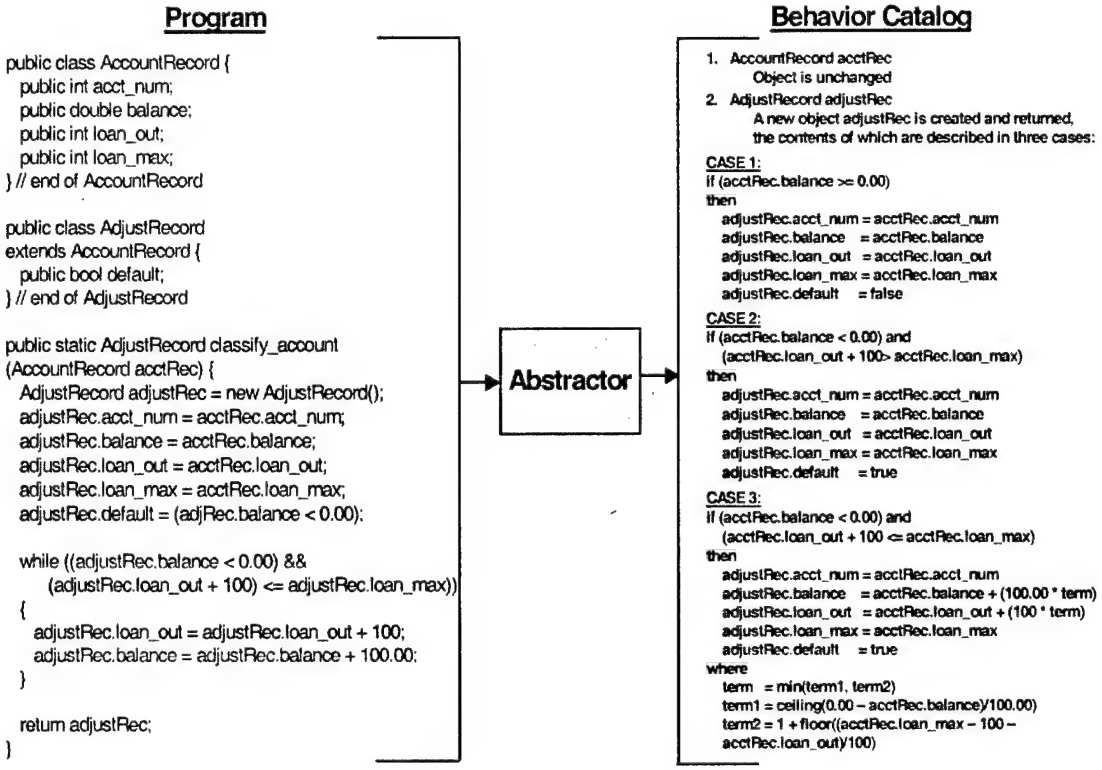


Figure 5: The Abstracted Behavior Catalog of a Java Program

Behavior calculation has potential for transformational impact on software and systems engineering through major reduction in effort combined with improved quality. Typical uses include:

- malicious code

  A security specialist periodically submits a critical operational program to an abstractor to determine if its behavior catalog reveals any malicious code recently inserted by programmers or intruders.

- error detection

  A quality assurance specialist submits a recently completed program to an abstractor and analyzes the calculated behavior catalog for incorrect behavior with respect to program requirements and specifications.

- new program development

  A software engineer periodically submits a partial program under development to an extractor to determine if the function defined by its behavior calculation is the function intended.

- just-in-time component composition

  A systems engineer employs an abstractor to generate behavior catalogs that guide rapid and reliable composition of components in responding to new system requirements.

- program maintenance

  A software engineer submits a program undergoing maintenance to an abstractor to determine where and how to make changes, and later resubmits it to ensure that changes have the desired effect.

- COTS product evaluation

  A systems engineer requests a product behavior catalog from a COTS software vendor to evaluate for planned use in a new system.

- reuse

  A systems engineer submits a program to an abstractor to generate its behavior catalog for evaluation of potential reuse in a new system.

- documentation

  A software engineer submits a completed program to an abstractor to generate its final behavior catalog for future maintenance and evolution.

If automated behavior extraction were easy, it would be commonplace by now. While much of the required mathematical theory already exists, this is nevertheless a hard problem with a number of challenges:

- loop function abstraction

  No general theory for loop abstraction exists. Because even a single while loop can compute an arbitrary partial recursive function, many results from the area of computability theory stand in the way. For example, the undecidability of the Halting Problem means that there will be some terminating while loops that the automated behavior abstraction system will not be able to detect as terminating. The undecidability of program function equivalence implies that an automated behavior abstraction system will need to use multiple representations of the same program function. The research approach here includes use of recursive expressions to represent loop operations as a starting point for behavior extraction, and development and application of canonical patterns and behavior templates for loops. Undecidability results from computability theory will be used to guide research choices along feasible directions. Potential limitations at the mathematical level can often be dealt with effectively at the engineering level to produce satisfactory solutions. This appears to be the case with respect to loop abstraction.

- expression simplification and reduction

  It is important to control the complexity of calculated behavior expressions as they propagate to higher levels. While much complexity reduction is intrinsic to function-

theoretic abstraction, research is required to investigate promising mathematical methods for unification and elimination of cases, as well as appropriate human factors engineering for effective display and analysis of behavior catalogs. A key component of this complexity reduction is the appropriate use of definitions to represent units of behavior that recur frequently throughout a program. Such use of definitions has long been applied in mathematics to render theorems and proofs more understandable. For example, although it is possible to do set theory using only "epsilon" (set membership) as the sole non-logical symbol, in practice it is impossible to express even the axioms of Zermelo-Fraenkel set theory in this manner without a complete loss of understandability. Facilities for specifying and integrating definitions into the process will be an important capability for an abstraction engine.

- indirect data references

  Many languages permit pointer references (either explicit or implicit) to data items that introduce a level of indirection that must be accommodated in the semantics of behavior calculation. The research approach here will include mapping of data references into canonical reference frameworks as a starting point for analysis. In particular, all objects are allocated on the heap in Java, so the problem of aliasing (i.e., different variable names referring to the same storage location) is substantial. Conversely, this is an area where automatic analysis can bring a significant amount of benefit to understanding the implications of the data layout of an unknown program.

The solutions to these challenges will enable abstraction engine development as a key enabler for the trustworthy systems of the future. In fact, it is difficult to imagine how trustworthy computing can ever be reliably achieved without knowing what programs do in all circumstances of use. In the current state of the art, this knowledge is sporadically and imperfectly accumulated from specifications, designs, code, and test results, all potentially incomplete and incorrect. Dynamic program modifications and compositions in modern network-centric systems severely limit the value and relevance of even this hard won but static and suspect knowledge. But programs are mathematical artifacts subject to mathematical analysis. Human fallibility still exists in interpreting the analytical results, but there can be little doubt that routine availability of calculated behavior would substantially reduce errors and vulnerabilities in software and make intrusion and compromise more difficult and detectable. Furthermore, questions about system trustworthiness capabilities for authentication, encryption, filtering, etc., are in large part questions about the behavior of programs that implement them. And because programs are subject to dynamic change and adaptation, only automated analysis can maintain the currency and relevance of behavior knowledge at little cost.

4 The Architecture of an Abstraction Engine

Figure 6 depicts a notional architecture of a program function abstractor. Denotational (functional) semantics are defined for the control and data structures of the target language, and possibly the machine, whose programs are to be abstracted, as well as for the forms of the behavior expressions that will represent the abstracted behavior. These semantics are stored in data repositories and employed to verify the correctness of the abstractor, to ensure that the calculated behavior indeed corresponds to the behavior of the program being abstracted. The behavior calculations are provided to a graphical interface to create presentation formats with appropriate human factors. Users need never be exposed to the underlying mathematics, but can have confidence in the abstracted behavior in the knowledge that it was derived with sound mathematical methods.

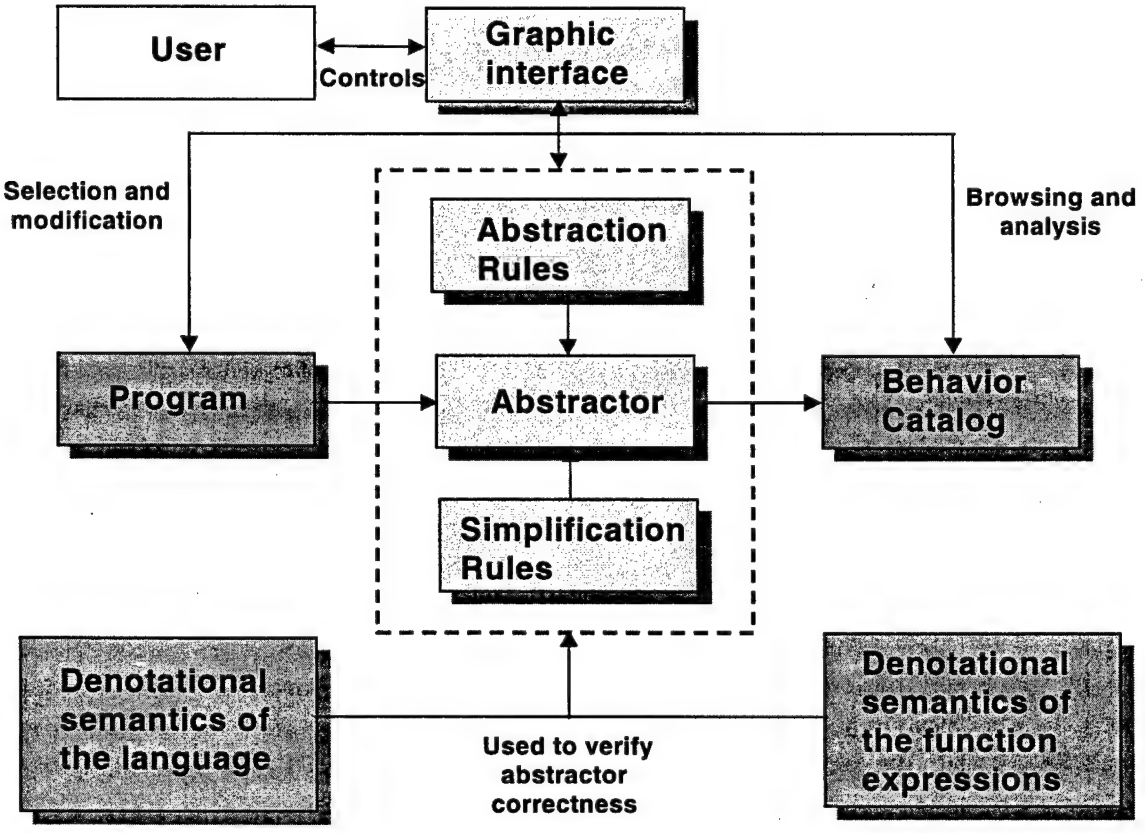


Figure 6: Architectural Structure of an Automatic Abstraction Engine

The abstractor itself employs abstraction and simplification rules to the stepwise derivation of program functions for each of the control structures of the input program or program part. The resulting behavior catalog is available for browsing and analysis.

A sensible, low-risk development process for such an abstractor would focus on creation of a prototype that addresses a subset of the target programming language, in order to gain early operational experience with the abstraction process. The subset would gradually be extended to eventually incorporate all language capabilities. As experience grows, new abstraction and simplification rules and techniques could be added in an iterative process of system development.

Abstraction engine development would require a small team of mathematicians and language semanticists with deep experience in function-theoretic mathematics and their software engineering application. The team would investigate and apply a spectrum of mathematical methods ranging from logic and fixpoint theory to model and category theory. Virtually all of the required mathematical foundations already exist. The principal task is to identify, specialize, and combine them for application to the behavior calculation problem in a process of iterative prototype evolution. There is no need to revisit the Halting Problem or deal with incomputable functions. A rich variety of concrete and computable mathematics is readily available. A reasonable approach would be to first identify foundations for abstracting sequential logic, then progress to concurrent logic. It is important to note that mathematical foundations are a means to an end, not an end in themselves. The overarching goal is practical automation of program behavior calculation and its routine engineering application.

5 Using Abstraction in Automated Verifiers, Integrators, and Certifiers

As noted above, automated function abstraction is unimaginable in today's state of the art. But if such a capability were developed, additional automation of traditionally manual and error-prone software engineering tasks would become possible, as discussed next.

5.1 Automated Program Verifiers

Function abstractions derived by automated abstractors are as-designed specifications of programs, and can be checked by software engineers in team inspections for correctness subject to human fallibility.

Beyond human inspection, it is possible to envision automated program verifiers that could compare function abstractions to intended specifications of programs for correctness with no human fallibility in CPU time scale. In this case, a program, its intended specification, or both may be correct or incorrect. Intended specifications must be defined by software engineers for this purpose. Verifier technology requires a capability for automated abstraction to derive as-designed specifications to be compared to intended specifications. Imagine a software engineer writing an intended specification and a corresponding refinement into a program part, then invoking a verifier on the fly to receive confirmation (or not) of correctness before continuing.

5.2 Automated Program Integrators

Consider the following miniature component and its as-designed function abstraction derived in a trace table:

```
do
  x := x + 2;
  y := x – y;
  x := y – x;
enddo
```

operation	x	y
x := x + 2	x1 = x0 + 2	y1 = y0
y := x – y	x2 = x1	y2 = x1 – y1
x := y – x	x3 = y2 – x2	y3 = y2

$$x3 = y2 - x2$$
$$= x1 - y1 - x1$$
$$= -y1$$
$$= -y0$$

$$y3 = y2$$
$$= x1 - y1$$
$$= x0 + 2 - y0$$
$$= x0 - y0 + 2$$

Thus, the function abstraction of the component is:

$$x, y := -y, x - y + 2$$

Now consider integrating this component as the second part of a sequence composition with the first part an exchange component as shown below:

```
do
   x, y := y, x;
   x, y := –y, x – y +2;
enddo
```

What is the net effect of this integration? This question can be answered by a trace table that calculates the net effect of the two components in sequence:

operation	x	y
x, y := y, x	x1 = y0	y1 = x0
x, y := –y, x – y +2	x2 = –y1	y2 = x1 – y1 + 2

$$x2 = -y1$$
$$= -x0$$

$$y2 = x1 - y1 + 2$$
$$= y0 - x0 + 2$$

Thus, the net effect of the integration is the following function abstraction:

$$x, y := -x, y - x + 2$$

Imagine a systems engineer attempting to integrate these components. Each component has an as-designed abstraction that can be furnished to an integrator to evaluate the composition and derive its net effect in CPU time scale. Components paired with corresponding as-designed abstractions are thereby enabled for such automated integration. Imagine changing a line of code in a large program. How does that change affect its function, correctness, and existing integration relationships with other components? With automated abstraction and integration technology these questions can be answered as routine engineering processes.

5.3 Automated Program Certifiers

Sizable programs exhibit virtually infinite populations of possible executions. It is an immutable fact of software engineering that no testing effort, no matter how extensive, can hope to exercise more than a minute fraction of these executions. So all testing is really sampling from an infinite population, and the only important question is how to draw the sample of test cases. If the sample is representative of eventual field use, then test results from the sample can be extrapolated to the entire population to make scientifically valid estimates of future field experience with all the executions that could not be tested. Such results are invaluable in making informed decisions on product quality and fitness for use, development process effectiveness, and resource allocation for testing.

This process is statistical usage-based testing. It requires as a starting point a definition of the population of possible executions augmented with estimated usage probabilities. This definition can be derived from the as-designed function abstraction of a system under test. So abstraction technology informs a scientific testing process that produces valid predictions of fitness for use of a system in a particular usage environment. In addition, as-designed abstractions can serve as oracles for evaluating the results of test executions. Imagine a systems engineer considering use of an abstraction-enabled component in a particular usage environment. The as-designed function abstraction can be furnished to an automated certifier along with a description of usage to produce valid predictions of how the component will perform. It is also important to note that verified programs have been determined to be correct with respect to their specifications with no testing or execution involved. It is this level of quality that permits the objective of testing to shift from attempting to improve quality through debugging, an impossible task, to focus on scientific certification of fitness for use.

Developing automated abstractors, verifiers, integrators, and certifiers will be difficult. The good news is that the most of the required theoretical foundations already exist. The place to start with is automated function abstraction, as the foundation for the other capabilities. The required work is substantial, but does not exceed the knowledge of how to do it.

6 Acknowledgements

The research work described in this report builds on Flow-Service-Quality engineering concepts, in particular the foundations of Flow Structures. It is a pleasure to acknowledge the contributions of Dr. Alan Hevner, Dr. Mark Pleszkoch, and Dr. Gwendolyn Walton to the development of those concepts. Thanks are also due to Dr. Richard Pethia and Dr. Thomas Longstaff, for sponsoring a visiting scientist in the CERT Coordination Center to explore foundations for automated function abstraction.

References

[Hausler 1990] Hausler, P.; Pleszkoch, M.; Linger, R.; & Hevner, A. "Using Function Abstraction to Understand Program Behavior." *IEEE Software 7*, 1 (January 1990): 55-63.

[Hevner 2001] Hevner, A.; Linger, R.; Sobel, A.; & Walton, G. "Specifying Large-Scale, Adaptive Systems with Flow-Service-Quality (FSQ) Objects," 110-120. *Proceedings of the 10th OOPSLA Workshop on Behavioral Semantics*. Tampa, Florida, October, 2001. New York, NY: ACM Press, 2001.

[Hevner 2002] Hevner, A.; Linger, R.; Sobel, A.; & Walton, G. "The Flow-Service-Quality Framework: Unified Engineering for Large-Scale, Adaptive Systems," *Proceedings of the 35th Annual Hawaii International Conference on System Science (HICSS35)*. Hawaii, Jan. 7-10, 2002. Los Alamitos, CA: IEEE Computer Society Press, 2002.

[Hoffman 2001] Hoffman, D. & Weiss, D., eds. *Software Fundamentals: Collected Papers by David L. Parnas*. Upper Saddle River, NJ: Addison Wesley, 2001.

[Linger 2002] Linger, R.; Pleszkoch, M.; Walton, G.; & Hevner, A. *Flow-Service-Quality Engineering: Foundations for Network System Analysis and Development* (CMU/SEI-2002-TN-019). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. <http://www.sei.cmu.edu/publications/documents/02.reports/02tn019.html>.

[Mills 1986] Mills, H.; Linger, R.; & Hevner, A. *Principles of Information System Analysis and Design.* San Diego, CA: Academic Press, 1986.

[Mills 2002] Mills, H. & Linger, R. "Cleanroom Software Engineering." *Encyclopedia of Software Engineering*, 2nd ed. New York, NY: John Wiley & Sons, Inc., 2002.

[Morgan 2002] Morgan, T. *Business Rules and Information Systems: Aligning IT with Business Objectives.* Reading, MA: Addison-Wesley, 2002.

[Parnas 1994] Parnas, D. & Wang, Y. "Simulating the Behavior of Software Modules by Trace Rewriting Systems." *IEEE Transactions on Software Engineering 19*, 10 (October 1994): 750-759.

[Pleszkoch 1990] Pleszkoch, M.; Hausler, P.; Hevner, A.; & Linger, R. "Function-Theoretic Principles of Program Understanding." *Proceedings of the 23rd Annual Hawaii International Conference on System Science (HICSS23).* Hawaii, January, 1990. Los Alamitos, CA: IEEE Computer Society Press, 1990.

[Prowell 1999] Prowell, S.; Trammell, C.; Linger, R.; & Poore, J. *Cleanroom Software Engineering: Technology and Practice.* Reading, MA: Addison Wesley, 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	January 2003	Final

4. TITLE AND SUBTITLE
Applying FSQ Engineering Foundations to Automated Calculation of Program Behavior

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Richard C. Linger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2003-TN-003

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

No software engineer can say with assurance how a sizable program, with its virtually infinite number of possible execution paths, will behave, that is, what it will do, in all circumstances of use. This incredible reality, widely acknowledged but little discussed, lies at the heart of intractable problems experienced in software development and use over the past 40 years. If full behavior is unknown, so too are embedded errors, vulnerabilities, and malicious code that can emerge in use. While this reality has seemed inevitable in the past, it need not be so in the future. The SEI CERT Coordination Center has been conducting research on Flow-Service-Quality (FSQ) engineering for complex, network-centric system analysis and development. FSQ Flow Structures treat the control structures of programs as rules, or implementations, of mathematical functions, that is, mappings from domains to ranges. The function, or behavior, of any control structure can be abstracted into a procedure-free statement that specifies its net functional effect in all circumstances of use with mathematical precision. The finite number of control structures in a program can be abstracted in stepwise fashion in an algebra of functions, to arrive at a precise statement of the program's overall behavior. The mathematical foundations largely exist, and development of such a capability is feasible, albeit difficult. Automated program behavior calculation would have a dramatic effect on software and systems engineering, and enable a new level of assurance in trustworthy systems. This report briefly summarizes research to date on Flow Structures and describes the application of their function-theoretic mathematical foundations to the problem of program behavior calculation.

14. SUBJECT TERMS	15. NUMBER OF PAGES
program understanding, program comprehension, program behavior calculation, function abstraction, function-theoretic methods, vulnerability detection, malicious code detection	34

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18 298-102